A Tool for Automated Inspection of Software Design Documents and Its Empirical Evaluation in an Aviation Industry Setting

This is the post-print of a paper that has been published with the following DOI: https://doi.org/10.1109/ICSTW.2016.12

A tool for automated inspection of software design documents and its empirical evaluation in an aviation industry setting

M. Evren Coşkun, M. Melta Ceylan, Kadir Yiğitözu
Turkish Aerospace Industries Inc. (TAI)
Ankara, Turkey
{ecoskun, mceylan, kyigitozu}@tai.com.tr

Vahid Garousi
Software Engineering Research Group, Department of Computer Engineering, Hacettepe University, Ankara, Turkey
vahid.garousi@hacettepe.edu.tr

***Abstract*-- While software inspection is an effective activity to detect defects early in the software development lifecycle, it is an effort-intensive and error-prone activity. Motivated by a real need in the context of the Turkish Aerospace Industries Inc. (TAI), a tool named AutoInspect was developed to (semi-) automate the inspection of software design documents and, as a result, to increase the efficiency and effectiveness of the inspection process. We present in this paper the features of the tool, its development details and its initial evaluation for inspecting the design documents of three real systems in the company. The results of the initial case-study reveal that the tool is indeed able to increase the inspections efficiency and effectiveness. In terms of efficiency, inspection engineers who used AutoInspect performed 41-50% more efficiently, for the three design documents under study, compared to the case when the tool was not used (i.e., manual inspections). In terms of effectiveness, compared to manual inspections, the automated approach found between 23-33% more defects in the three design documents under study. As the tool currently only provides partial automation, our efforts are currently underway to increase its automation level even further.**

Keywords*-- *Software inspections; design verification; automated inspections; industry case study; improving efficiency and effectiveness; Computer-aided software engineering (CASE).

I. INTRODUCTION

Software inspection is a detailed review of software artifacts by technically-competent peers (defined and used as early as in 1976 [1]). Inspection is considered an efficient and effective means of defect detection in software engineering. The success of inspection is due to its early defect detection capability, when the cost of defect removal is less, compared to later phases of software development lifecycle [2].

The software engineering literature contains many sources on software inspections as an important activity. For example, the book by Gilb and Graham [3] in 1993 is one of the early books on this topic, and cites numerous positive experience reports on the topic. For example, Russell reported [4] a return of 33 hours of maintenance saved for every hour of inspection invested in a case study of inspections of 2.5 million lines of high-level code at Bell-Northern Research. Furthermore, Barry Boehm included inspection (phrasing it as "walkthroughs") in his list of the 10 most important approaches for improving software quality because, according to his research, it could catch 60% percent of defect [5]. Inspections are conducted in different phases of the development lifecycle and on different software artifacts (e.g., requirements and design documents, source code, and test scripts) [6].

While inspection is an effective activity to detect defects, it still remains an effort-intensive and error-prone activity [3]. Inspectors (also called, inspection engineers) have to spend many hours of manual effort to look for potential defects in software artifacts, usually using pre-specified checklists or inspection rules. While the goal of the activity is finding potential defects, the activity itself can unfortunately be error-prone since it is mostly conducted by humans and, thus, even by following high-quality checklists, inspector can miss potential defects.

In the context of the Turkish Aerospace Industries Inc. (TAI), the authors and their colleagues were facing the above challenges in the scope of inspection activities on software design documents for the company's Enterprise Resource Planning (ERP) software applications. Motivated by that need, we started a project to explore ways to increase the efficiency and effectiveness of the inspection process. The project was based on the principles of the Action-Research (AR) methodology [7] in which the real industrial challenges (problems) drive the research. As the result of this project so far, we have developed a tool named AutoInspect to (semi-) automate the inspection of software design documents and, as a result, to increase the efficiency and effectiveness of the inspection process.

The remainder of this paper is structured as follows. We describe the case, company context, and the need for the proposed tool in Section 2. A review of the related work and tools is presented in Section 3. Section 4 presents features, example usage, and development details of the AutoInspect tool. Section 5 presents a preliminary industrial case-study for evaluation of the tool. Finally, in Section 6, we draw conclusions, and discuss our ongoing and future works.

II. CASE DESCRIPTION AND NEED ANALYSIS

A. Company and context

Turkish Aerospace Industries Inc. (TAI) is a major center of technology in design, development, manufacturing, integration, modernization and after-sales support of aerospace systems in Turkey. TAI's experience includes the licensed production of F-16 Fighting Falcon jets, combat search and rescue (CSAR) and utility helicopters as well as the design and development of Unmanned Aerial Vehicles (UAVs), fixed and rotary wing aircrafts. TAI's core business also includes modernization, modification and systems integration programs and aftersales support of both military and commercial aircrafts.

One of the departments of the company is the Information Management Systems (IMS) department that develops in-house Enterprise Resource Planning (ERP) applications and conducts all the related analysis, design, development, testing, and the process improvement activities. The department develops the ERP applications using the industry's best practices, and these applications support all business functions of the company. The IMS department is responsible to develop and maintain applications for 12 main process areas of the company: (1) information and knowledge management, (2) finance management, (3) human resources management, (4) product development, (5) production planning, (6) manufacturing execution, (7) quality management, (8) facility asset management, (9) purchasing and subcontract management, (10) sales & transportation management, (11) logistics management and (12) portfolio management.

When developing software systems, as per the development process followed in the company, software design documents are developed as a result of the analysis and design phases during the software development lifecycle. Design documents are prepared by the engineers in the 'Business Process' group of the IMS department in accordance with the department's pre-designed software design document template and guidelines. The documents mainly include the following artifacts: business flow diagrams, graphical user interfaces design, functional/non-functional requirements, special hardware requirements, constraints, database design, integration with other systems, and information/error messages. Figure 1 shows an example page from an actual design document which undergo inspections.

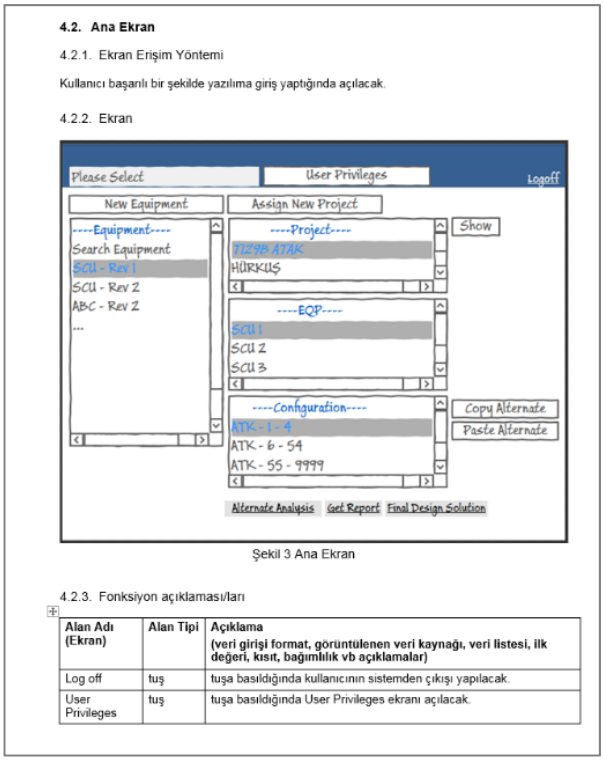

FIGURE 1-AN EXAMPLE PAGE FROM AN ACTUAL DESIGN DOCUMENT WHICH UNDERGO INSPECTIONS

B. *Inspection of software design documents: the traditional manual approach*

After software design documents are developed, again as per the company's development process, the Software Verification and Validation group of the IMS department is tasked to manually inspect the design documents according to a specific design inspection (verification) checklist, as shown in Table 1. Besides the checklist, for the purpose of manual inspections, software engineers also use the software design document template, the database design guide, the requirements management instruction and the department's software verification and validation instructions. As the result of the inspection, non-compliances are recorded in the department's Application Lifecycle Management (ALM) tool, i.e., Microsoft Team Foundation Server (TFS), and they are later assigned to the developer of the document to be fixed. The inspection process is continued until all issues are resolved and software development cycle continues afterwards, i.e., the implementation phase.

The 20 rules of the inspection checklist, shown in Table 1, are divided into two groups: structure and contents. The first two inspection rules are about document structure. For each rule, we also show in Table 1 the level of automation that we have achieved so far by the AutoInspect tool, to be presented in the rest of this paper. Also, for each rule, the severity of possible incompatibilities that the inspector can choose form is shown, e.g., for rule #1, the possible values are {low, medium, high} while for rule #2, it can only be 'High', if any.

As we discuss in Section 4, to automate some of the rules in the manual checklist in Table 1, we had to break them down into sub-rules when implementing our tool (AutoInspect). Thus, the column named '# of sub-rules in AutoInspect' in Table 1 shows that information, and the issue will be discussed in detail in Section 4.2.

Also, another important aspect in Table 1 is the level of automation by the AutoInspect tool which can be either: {No, Partial, Full}. According to a survey paper [8] on inspections, there are four main areas of inspections which have been the target for tool support (automation): document handling, individual preparation, meeting support and metrics collection. For the rules in Table 1 which have 'partial' levels of automation, the tool provides the following features: (1) document handling: by automatically browsing the design document and showing the exact location to be inspected to the inspection engineer, (2) metrics collection: by automatically collecting the number of defects and placing them in an output report, and (3) coordination of and offering a collaborative inspection approach in which the tool and inspection engineer work together to conduct the inspection activities. These activities will be discussed later in the paper.

C. *Inefficiency of manual inspection and need for automated inspection*

As discussed above, similar to many other companies [2], manual inspection process has been carried out in the group for several years now in which inspection engineers were verifying the documents manually with respect to the design checklist. However, similar to every manual task and process, manual inspections were error-prone, ineffective, and inefficient in

several ways: (1) as per the time logs, manual inspection of the 109-page design document for a system called EFAB (acronym in Turkish for: "*Elektronik Fabrika Sistemi*", meaning: Electronic Factory System) took about 29 man-hours; (2) since there are various checklists to be controlled, inspectors could easily miss some of the rules and, thus, defects would stay in the documents; (3) the inspection process was not streamlined and were somewhat ad-hoc; (4) the process was lacking the adequate traceability, visibility and reliability, e.g., non-compliance items did not have enough explanations to locate the exact location of the issues in document.

III. RELATED WORK AND TOOLS

When searching for "automated software inspection" in academic search engines, e.g., Google Scholar and Scopus, one would find a large number of papers, most of which are related to inspection of code (e.g., static code analysis tools) [9]. To stay relevant on our study scope, we only narrowed our literature search and review to studies on and tools used for inspection of technical documents, e.g., design and requirements documents. The other relevant set of keywords that we searched for was "software design verification". Among the studies that we found are [10-14].

TABLE 1- MANUAL CHECKLIST (SET OF INSPECTION RULES) USED FOR MANUAL INSPECTION OF DESIGN DOCUMENTS

Group	Inspection rules (criteria)	Severity of possible incompatibilities				Level of automation by the AutoInspect tool	# of sub-rules in AutoInspect
		Low	Medium	High	Critical		
Structure compliance	1. Is the document format compatible with the software design document template? (itself has a set of rules, examples in Section 4.2)	X	X	X		Full	20
	2. Are the versions of the design document in the TFS server same as the one on the SharePoint server?			X		No	0
Contents compliance	3. Are the functional requirements defined?				X	Partial	1
	4. Are the user interfaces and their features defined?	X	X	X	X	Partial	3
	5. Is the user interface flow clear and stated properly?	X	X	X	X	Partial	4
	6. Are decision states, elections / queries and calculations clearly defined?	X	X	X	X	Partial	2
	7. Are all requirements classified and enumerated?		X			Partial	1
	8. Is the database design compatible with the database guide?		X	X	X	Partial	24
	9. Are all user groups and their behaviors clearly defined?			X	X	Partial	2
	10. Are all information security requirements clearly defined?				X	Partial	2
	11. Are all hardware requirements clearly defined?		X	X	X	Partial	1
	12. Are all information/error messages, and in which case they are shown, defined?		X	X		Partial	3
	13. Is there any requirement which is out of the scope? (Except those written as a note)		X	X		Partial	1
	14. Is the integration with other systems defined?			X	X	Partial	1
	15. Are there any conflicting or inconsistent requirements?		X	X	X	Partial	3
	16. Do error messages clearly indicate what action the user needs to take to correct the error?	X	X	X		Partial	2
	17. Are assumptions and limitations stated?	X	X	X	X	Partial	1
	18. Are the requirements clearly understood?	X	X	X	X	Partial	1
	19. Can the requirements be tested? (Dependent on the test/hardware tools, the test methods, the test resources, the scalability and the observability.)	X	X	X		Partial	1
	20. Are the requirements traceability satisfied via TFS?			X		No	0

By considering the above deficiencies of the manual inspection process and upon the review of the current inspection process by the team members and managers, the team decided to switch to an automatic inspection process. For this, we had to find, adopt, customize or develop (from scratch) an automatic inspection tool. Based on the AR methodology [7], the first task after identifying the need was to review the related work and tools. The review of the state-of-the-art and -practice would actually serve two purposes: (1) to see if we could find an existing commercial or open-source inspection tool which we could customize/adopt to our need; and (2) to become familiar with the type of automated inspection approaches/methods proposed by other researchers and practitioners.

The paper [10] presented a tool called QuARS (Quality Analyzer of Requirements Specification) for the analysis of natural language software requirements. The tool is based on a special quality model which aims at providing a quantitative, corrective and repeatable evaluation of software requirement documents.

The paper [11] presented an early lifecycle tool for assessing requirements in natural language developed by NASA's Goddard Space Flight Center's (GSFC). The tool searches the document for terms identified as 'weak' phrases. The reports produced by the tool are used to identify specification statements and structural areas of the requirements specification document that need to be improved. The metrics can be used by project managers to recognize and preclude potential areas of risk. Similarly, [12] presented a tool named *Text2Test* for automated inspection of

natural-language use cases. The experience paper [13] reported the in-process inspections of design and development work products at AT&T. Brykczynski reported in 1999 paper [14] a survey of software inspection checklists.

Software documentation quality is also another active related field to our work, e.g., works such as [15-18]. The work in [17] presents a quality monitoring method for the automated quality assessment of software documentation using a document quality analysis framework and a set of quality rules which represent best practices for software documentation. To shows the value of software documentation quality, [18] conducted a survey software professional and the survey shows that the most important quality attributes with regard to documentation quality are accuracy, clarity, consistency, readability, structuredness, and understandability. Many respondents mentioned a general lack of tool support for quality assessment of software documentation.

After reviewing the related work and tools, we really did not find any available tool suitable for our needs. Thus we decided to develop our own inspection tool.

IV. AUTO INSPECT: FEATURES AND DEVELOPMENT DETAILS

After deciding to develop our own inspection tool, several engineers in the team were tasked to design and develop the tool. The goal was to develop a reliable, efficient, flexible, extendible and user-friendly tool. We discuss next the features, example usage, and implementation details of the tool.

A. *Features and example usage*

AutoInspect is a tool for semi-automated inspection of design documents. The activity diagram in Figure 4 shows the usage process and input/outputs of the tool. A software design document is given by the engineer as the input. Similar to other inspection tools, e.g., [10-13], our tool does not fully automate inspections, but only partially automates it. As a 'collaborative' inspection tool, the engineer 'interacts' with the tool and the tool facilitates and increases the inspection efficiency in the tasks that our experience has shown to be tedious and effort intensive, e.g., browsing through a long design Word document and checking formats, etc. The other important input is the set of design inspection rules, examples of which were provided in Table 1. As outputs, a list of defects found during inspection, and a design verification report (in PDF format) are created. We discuss the details with examples in the following.

As discussed in Section 2, the inspection rules are divided into two groups: structure and contents of the documents. Accordingly, the tool's usage follows the same notion. Figure 2 shows a screenshot of the tool's main windows also the results of a given document's structure inspection. The two button's "Inspect Structure" and "Inspect Contents" conduct the two main use cases. It is imperative to execute the structure inspection first and then the content inspection. To ensure flexibility and extensibility of the tool, the inspection rules are stored out of the tool's core code-base and can be revised/extended as needed (to be discussed in the next sub-section).

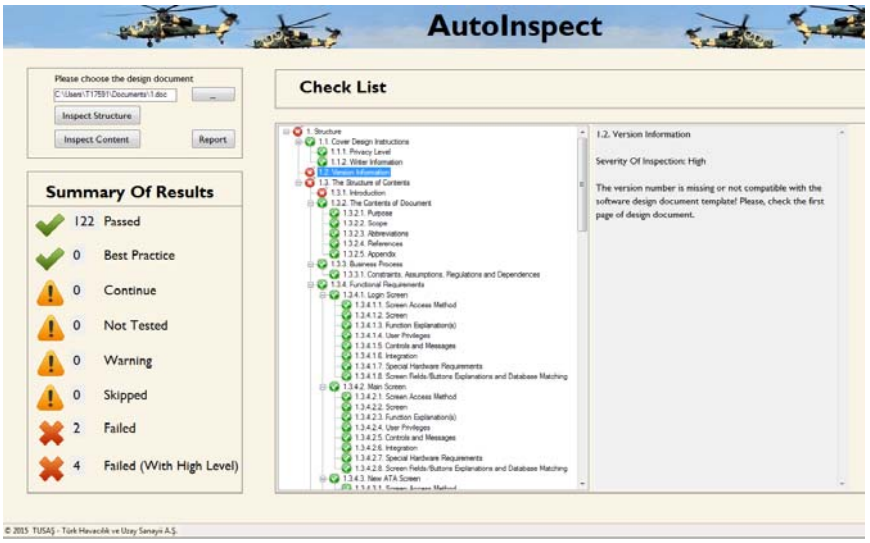

FIGURE 2-A SCREENSHOT OF THE TOOL SHOWING THE RESULTS OF A DOCUMENT'S STRUCTURE INSPECTION

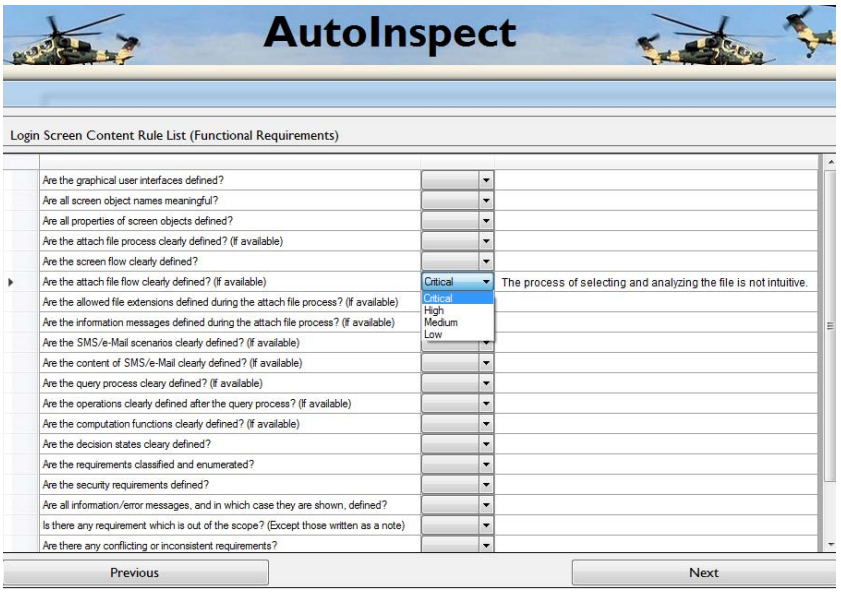

FIGURE 3- A SCREENSHOT OF THE TOOL SHOWING THE INTERACTION WITH THE ENGINEER TO CONDUCT SEMI-AUTOMATED INSPECTION OF DOCUMENT CONTENTS

During the structure inspection, the rules are checked automatically such as the required contents of design documents, version information, contents of GUI elements, tables and figures, and database design diagrams. Summary of result, as shown in Figure 2, is shown to the user after the structure inspection is completed. In this example, AutoInspect has been applied to a design document for a system named EYTS. After the structure inspection, in this case, the tool was able to find two non-compliances (defects) in this case. Further explanations of the non-compliances can be seen by pressing the red icons, an

example of which is also shown in Figure 2. In this case, the document under inspection is missing the version information and text inside the introduction section.

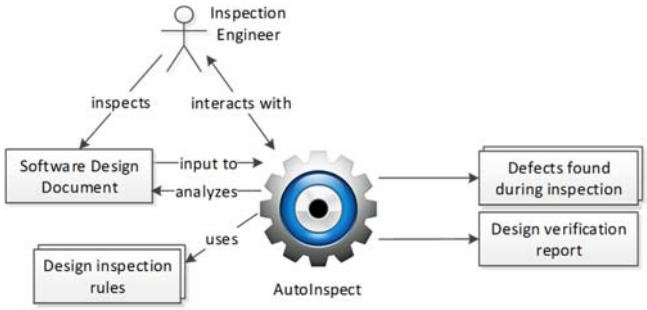

FIGURE 4-ACTIVITY DIAGRAM SHOWING THE USAGE PROCESS AND INPUT/OUTPUTS OF THE TOOL

The tree structure of the checklist shown in Figure 2 is based on the design document template that company has created a few years ago and shall be used in all projects. After the initial parts, the major structure of a given document is about 'functional requirements' which shall be specified for each GUI screen for the system under design.

Once structure inspection is completed, the engineer will follow the process with content inspection using the tool. As discussed in Section 2.2 and shown in Table 1, content inspection has not been fully automated yet since expert human knowledge is still necessary for all the tasks in this category which can hardly be automated (that will require AI and other advanced machine learning techniques which we plan to pursue in future). Figure 3 shows a screenshot of the tool interacting with the engineer to conduct semi-automated inspection of document contents, in this case the functional requirements of a 'Login' screen. The content inspection phase is based on a relatively large set of inspection rules, examples of which are shown in Figure 3, e.g., (1) Are all object names meaningful?, and (2) Are the screen flows clearly defined?

TABLE 2- THE LIST OF INSPECTION SUB-RULES FOR MANUAL INSPECTION RULES WHICH HAD TO BE BROKEN DOWN INTO SUB-RULES IN AUTOINSPECT

	#*	Inspection rules
Structure	1	1- Is the version information written?
		2- Are the list of tables and the tables in the list defined?
		…
		20- Is the database design described?
Content	4	1- Are the graphical user interfaces defined?
		2- Are all screen object names meaningful?
		3- Are all properties of screen objects defined?
	5	1- Is the attach file process clearly defined? (If available)
		2- Is the screen flow clearly defined?
		3- Are the SMS/e-Mail scenarios clearly defined? (If available)
		4- Is the query process clearly defined? (If available)
	6	1- Are the computation functions clearly defined? (If available)
		2- Are the decision states clearly defined?
	8	1- Are the primary keys defined for all database tables?
		2- Is the datatype of all table columns defined?
		…
		22- Are the privileges defined for database tables?
		23 - Is the GUI object compatible with DDL?
		24- Is there any GUI object connected with the database column which is not defined in DDL?
	9	1- Are the user privileges defined for the entire software?
		2- Are the user groups and their behaviors defined for every single GUI?
	10	1- Are the information security requirements defined in the non-functional requirements section of the design document?
		2- Are the information security requirements defined for the GUI?
	12	1- Are the information messages defined during the attach file process? (If available)
		2- Are information/error messages, and in which case they are shown, defined in the function explanations section of the design document??
		3- Are information/error messages, and in which case they are shown, defined in the controls and messages section of the design document?
	15	1- Are there any conflicting or inconsistent requirements?
		2- Is there any conflicted definition for the same GUI objects which are stated in different screens?
		3- Is there any conflicted function for the same events which are stated in different screens?
	16	Is the content of information/error messages consistent?
		Is the content of information/error messages meaningful?

*: Rule number in Table 1

Furthermore, in terms of usability and to increase efficiency, the tool assists the inspection engineer by automatically jumping to the relevant part of the design document in the Microsoft Word which is opened automatically by the tool in a second display (monitor), usually to the right of the AutoInspect window. In this scenario, if the engineer notices any issue (non-compliance), s/he would select the severity of inspection and would type an explanation as well (if needed). The inspection engineer would then press the 'Next' button to check the next set of inspection rules. Summary of the results, shown in the left-side of Figure 2, is constantly updated, in each step of the inspection.

Once all the inspections are done, by pressing the 'Report' button, the tool generates a design verification report (in PDF format) as shown in Figure 5. In this example, after the structure and content inspections, a total number of 571 inspection rules have been applied, out of which 563 rules have passed, and the collaborative work of the tool and engineer has detected 8 issues (and 14 high-level defects in the tree structure).

FIGURE 5-A SCREENSHOT OF THE TOOL

B. *Inspection rules and sub-rules*

As discussed in Section 2.2, to make some of the rules in the manual checklist in Table 1 automatable or to make them concrete enough for the inspection engineer to atomically decide about them, we had to break them down into sub-rules when implementing our AutoInspect tool in or der to make them, e.g., rule #1 of the document structure in Table 1 (*Is the document format compatible with the software design document template?*) was broken down into 20 sub-rules, as listed in Table 2, which shows the list of all sub-rules for rules having more than one sub-rule. Manual inspection rules such as rule #3 in Table 1 (*Are the functional requirements defined?*) were concrete enough to be easily implemented in AutoInspect and thus there was no need to be broken down into sub-rules.

C. *Development details*

The team followed the iterative development process in which expert inspectors (who has expertise in manual inspection for a few years) iteratively worked with developers to develop the features one by one. Tool requirements were not formally written down, but instead, were informally communicated among the team members. Essentially, the senior engineers who had expertise in conducting the manual inspections transferred their knowledge to the developer to develop the tool.

Since design documents were all in Microsoft Word format, for compatibility and easier implementation purposes, we selected the Microsoft Visual Studio 2013 and C# as the development platform for AutoInspect. Moreover, we used suitable libraries such as `Microsoft.Office.Interop.Word` since this library allows Word DOC files to be easily opened and manipulated programmatically from C#.

To present further development details about the tool, Figure 6 shows the architectural design of AutoInspect. Each class, its purpose, and the meaning of the most significant attributes and functions are described next. The `MainScreen` class is the main GUI class of AutoInspect and its role is to get input from the inspection engineer, to create `InspectionManager` class (described next), to call structural and content inspection functions and to show results to the inspection engineer. The `ContentInspectionScreen` class is used to get inputs for the result of content inspection classes from the inspection engineer. The inspection engineer can select the severity of inspection and enter comments if needed. Moreover, the `ReportCreator` class is used to create final result report.

The `InspectionManager` class is the main operational class of AutoInspect and its role is to create `InspectionFactory` class (described next), to run structural inspection, and to return the inspection list with results to `MainScreen` class. The `InspectionFactory` class is used to create `StructureInspectionFactory` and `ContentInspectionFactory`. During the implementation, in order to make future code maintenance easier [19], we used the factory design pattern. These factory classes are used to create the `InspectionAbsract` classes, which is the analysis core of AutoInspect and all the inspection rule classes are inherited from this abstract class such as `StructureInspectionFind`, `StructureInspection-FindBetween`, `StructureInspectionFindHeader`, `StructureInspectionFindTable`, and `Content Inspection`.

The `inspect()` method is the most important function for the `InspectionAbstract` class. The results of this function are shown to the inspection engineer by the tool. Figure 7 shows the `AbstractIncpection` class can have `childIncpections`. If the inspection class has child inspections, it recursively calls the inspect function of `childIncpections` until there is no child inspections. If the returned valued of the `inspect()` method is false, the result of all high level inspect functions are false. For the structural inspection classes, the result of the `inspect()` method is automatically calculated. On the other hand, for the content inspection classes, the result of the `inspect()` method is entered by the inspection engineer and processed (aggregated) later by the tool.

V. INITIAL EVALUATION OF THE TOOL

A. *Case-study design*

Since the tool was developed based on a real need (as discussed in Section 2), once the tool development finished, we started evaluating the tool on a set of design documents to assess how it addressed the need behind it. In the initial step as we report in this tool paper, we started with a small case-study.

The research approach we used in our study is the Goal, Question, Metric (GQM) methodology [20]. Stated using the GQM's goal template [20], the goal of the case study is to evaluate the AutoInspect tool in increasing the efficiency and effectiveness of inspections when compared to manual inspections before the tool existed. To address the goal, we raised and addressed the following research question (RQ): To what extent the tool increases the efficiency and effectiveness of inspections when compared to manual inspections? As metrics to assess efficiency and effectiveness, we selected the inspection effort (in hours) and the number of defects found, respectively.

As the objects of study, we selected the design document of three ongoing projects (systems) in the company: (1) EYTS (acronym in Turkish for: "*Ekipman Yerleşim Tasarımı Sistemi*", meaning: Equipment Location Design System), (2) EFAB (acronym in Turkish for: "*Elektronik Fabrika Sistemi*", meaning: Electronic Factory System), and (3) TAS (acronym in Turkish for: "*TAI Akademi Sistemi*", meaning: TAI Academy System).

As the subjects of study, to collect the metrics for each object under study, we ensured that an engineer different than the one who had done manual inspection would conduct automated inspection using the tool. Also, to ensure comparability and to really assess the benefit of the tool and not the personal skill-level

nor efficiency and effectiveness of the inspector, we ensure that the two engineers had similar expertise and performance.

B. Results

Table 3 shows the size metrics of the three objects of study (design documents) and measurements as the results of the study. In terms of size metrics, the document had between 52-321 pages in the standard design template format and between 12-34 GUI screens.

In terms of efficiency, manual inspection efforts were collected from the time log records and, as shown in Table 3, are 8, 29 and 51 man-hours for the three documents. Automated inspection effort, on the other hand, varied between 4-27 hours. This, in turn, yielded efficiency improvements between 41-50% for the three cases. Automation has helped us save about 50% of the effort, and not more. The main reason for this is that the tool only mainly helps the inspector navigate the document and save the issues and does not do many sophisticated automated analysis.

In terms of effectiveness, compared to manual inspections, the automated approach found between 23-33% more defects for the three cases. In other words, the automated tool found all the defects, found by manual inspections, plus some additional defects. The main reason for this was that, as shown in Table 1, the high-level manual inspections rules had to be broken down to more granular lower-level automated inspections rules. As a result of rules' granularity and a decrease in their ambiguity, the defect detection effectiveness was increased.

Thus, in summary we clearly observe that our tool has been beneficial to the team, both in terms of efficiency and effectiveness.

C. Discussions, limitations and implications

As discussed in Section II, our semi-automated tool provided tool support for the following areas: (1) document handling: by automatically browsing the design document and showing the exact location to be inspected to the inspection engineer, (2) metrics collection: by automatically collecting the number of defects and placing them in an output report (see the example in Figure 5), and (3) coordination of and offering a collaborative inspection approach in which the tool and inspection engineer work together to conduct the inspection activities. The 'collaborative software inspection' in a notion that has actually been around since 2-3 decades ago, e.g., [8].

This is the post-print of a paper that has been published with the following DOI: https://doi.org/10.1109/ICSTW.2016.12

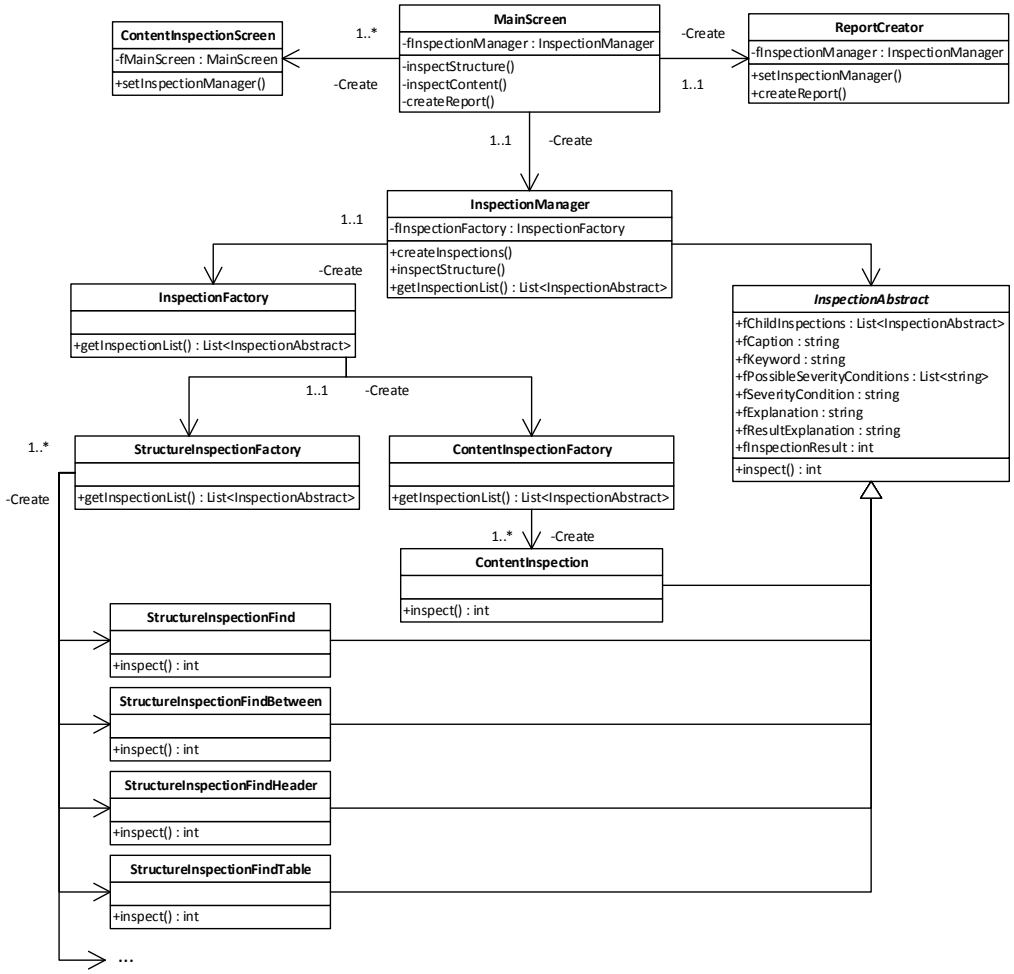

FIGURE 6-CLASS DIAGRAM SHOWING THE ARCHITECTURAL DESIGN OF THE TOOL AND USAGE OF THE FACTORY DESIGN PATTERN

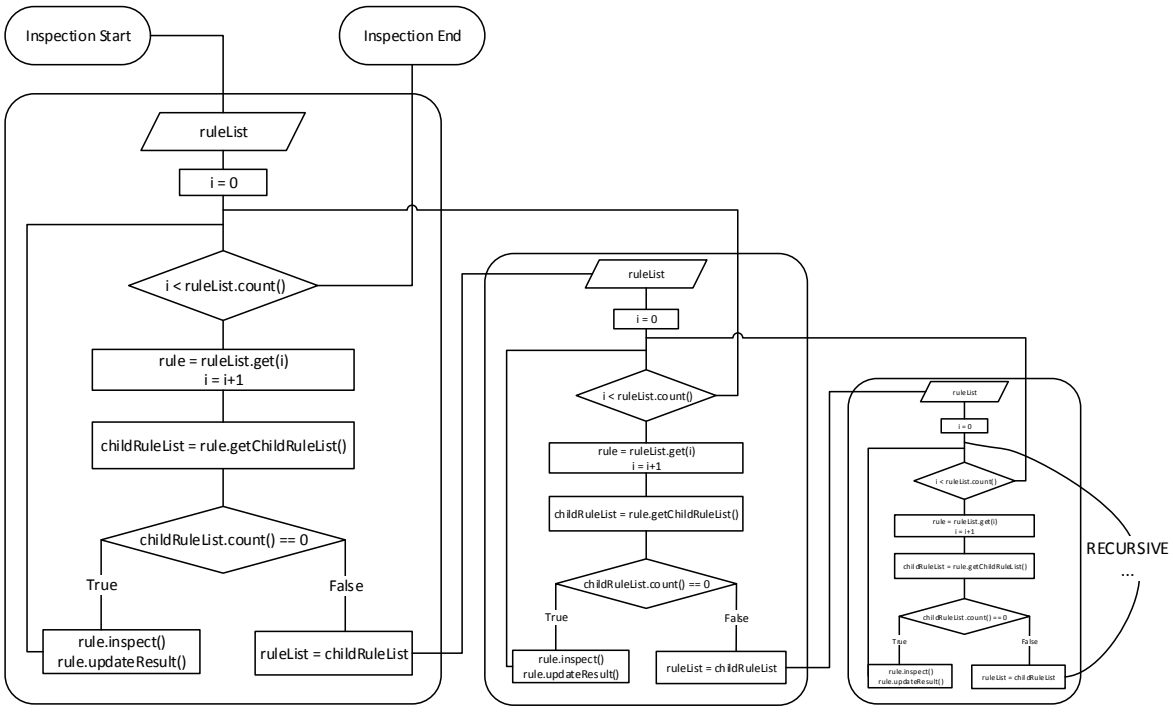

FIGURE 7-IMPLEMENTATION DETAILS OF THE `INSPECT()` METHOD

This is the post-print of a paper that has been published with the following DOI: https://doi.org/10.1109/ICSTW.2016.12

TABLE 3- RESULTS OF THE CASE STUDY

Design document name	Size metrics		Efficiency (inspection effort in hours)			Effectiveness (# of defects found)		
	Number of pages	Number of GUI screens	Manual	Automated	Improvement %	Manual	Automated	Improvement %
EYTS	52	12	8	4	50%	6	8	33%
EFAB	109	31	29	17	41%	100	129	29%
TAS	321	34	51	27	47%	13	16	23%

In terms of limitations of our approach, we are aware that the tool is not a fully automated approach yet since automated inspection of document contents (semantic aspects), for example, requires sophisticated techniques (e.g., based on artificial intelligence or and other advanced machine learning) and is currently done based on expert human knowledge. But still in this mostly manual task, the tool provides an interactive and collaborative help, thus it is still helpful.

In terms of the implications of our tool and its evaluation study, we believe that more work is needed by the software industry and the research community towards developing more automated tools in support of inspection since it is an effort-intensive and error-prone activity.

VI. CONCLUSIONS AND FUTURE WORKS

Motivated by a real need in the context of the Turkish Aerospace Industries Inc. (TAI) to increase the efficiency and effectiveness of inspection activities, a tool named AutoInspect was developed to (semi-) automate the inspection of software design documents. We presented in this paper the features of the tool, its development details and its initial evaluation for inspecting the design documents of three real systems in the company. The results of the initial case-study revealed that the tool is indeed able to increase the inspections' efficiency and effectiveness. We provided quantitative measurements to demonstrate that the tool increased efficiency and effectiveness of inspection activities in our team. However, due to confidentiality reasons, we could not provide more technical details about our approach and tool, we think that other practitioners may be able to adopt some of these ideas in developing their own automated tools and approaches.

Among our future work directions are: (1) integration of the tool with the issue management system used in the company, i.e., Microsoft Team Foundation Server (TFS); (2) as the tool currently only provides partial automation, our efforts are currently underway to increase its automation level; and (3) to empirical assess scalability and usability of the tool, i.e., to what extent does the tool scale to large sets of documents? And to what extent do the users find the tool usable?

ACKNOWLEDGEMENTS

The authors would like to thank the following internship students who were involved in the tool's development phase: Anıl Araç, Burak Kaan Bilgehan, Özlem Ceviz, Tolga Karaman, Uğur Konar, Özkan Mert Öztürk, and Osman Tayfun Yelim.

REFERENCES

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal,* vol. 38, pp. 258-287, 1976.

[2] J.-L. Boulanger, *Static Analysis of Software: The Abstract Interpretation*: John Wiley & Sons, 2013.

[3] T. Gilb and D. Graham, *Software Inspection*: Addison-Wesley, 1993.

[4] G. W. Russell, "Experience with inspection in ultralarge-scale development," *IEEE Software,* vol. 8, pp. 25-31, 1991.

[5] B. Boehm, "Industrial Software Metrics Top 10 List," *IEEE Software,* pp. 84-85, 1987.

[6] O. Laitenberger and J.-M. DeBaud, "An encompassing life-cycle centric survey of software inspection," *Journal of Systems and Software,* vol. 50, pp. 5-31, 1// 2000.

[7] P. S. M. d. Santos and G. H. Travassos, "Action research use in software engineering: An initial survey," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 414-417.

[8] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood, "Automating the software inspection process," *Automated Software Engineering,* vol. 3, pp. 193-218, 1996/08/01 1996.

[9] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, "Preliminary Results On Using Static Analysis Tools For Software Inspection," Proceedings of International Symposium on Software Reliability Engineering, 2004.

[10] F.Fabbrini, M.Fusani, S.Gnesi, and G.Lami, "An Automatic Quality Evaluation for Natural Language Requirements," in *Proceedings of the International Workshop on Requirements Engineering: Foundation for Software Quality*, 2001.

[11] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt, "Automated analysis of requirement specifications," Proceedings of the International Conference on Software Engineering, 1997.

[12] A. Sinha, S. M. Sutton, and A. Paradkar, "Text2Test: Automated Inspection of Natural Language Use Cases," in *International Conference on Software Testing, Verification and Validation*, 2010, pp. 155-164.

[13] P. J. Fowler, "In-process inspections of workproducts at AT&T," *AT&T Technical Journal,* vol. 65, pp. 102-112, 1986.

[14] B. Brykczynski, "A survey of software inspection checklists," *SIGSOFT Softw. Eng. Notes,* vol. 24, p. 82, 1999.

[15] J. Zhi, V. Garousi, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe, "Cost, Benefits and Quality of Software Development Documentation: A Systematic Mapping," *Journal of Systems and Software,* vol. 99, pp. 175–198, 2015.

[16] G. Garousi, V. Garousi, G. Ruhe, J. Zhi, M. Moussavi, and B. Smith, "Usage and Usefulness of Technical Software Documentation: An Industrial Case Study," *Elsevier Journal of Information and Software Technology,* vol. 57, pp. 664-682, 2015.

[17] A. Dautovic, "Automatic assessment of software documentation quality," in *International Conference on Automated Software Engineering (ASE)*, 2011, pp. 665-669.

[18] R. Plesch, A. Dautovic, and M. Saft, "The Value of Software Documentation Quality," in *International Conference on Quality Software (QSIC)*, 2014, pp. 333-342.

[19] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta, "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *Software Engineering, IEEE Transactions on,* vol. 27, pp. 1134-1144, 2001.

[20] V. R. Basili, "Software modeling and measurement: the Goal/Question/Metric paradigm," Technical Report, University of Maryland at College Park1992.